INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 756

PORTAGE D'UN ENVIRONNEMENT UNIX SUR LE NOYAU GOTHIC

Pascale LE CERTEN
Béatrice MICHEL
Gilles MULLER

NOVEMBRE 1987

PORTAGE D'UN ENVIRONNEMENT UNIX SUR LE NOYAU GOTHIC*

Pascale LE CERTEN, Béatrice MICHEL
BULL RECHERCHE

Gilles MULLER
IRISA-INRIA

Campus de Beaulieu, 35042 Rennes-CEDEX

Abstract :

Creating an operating system on an open machine, implies the use of development methods. This report describes the way chosen to implement the kernel of the GOTHIC distributed system and a well suited programming environment. Our goal is to design a development system which can run on the successive versions of the kernel. The major advantage of that method is to intensively test the kernel for programming and design errors.

Keys words :

Distributed systems, Unix.

Résumé :

La réalisation d'un système d'exploitation sur une machine nue impose l'utilisation de méthodes de développement. Ce rapport décrit la démarche choisie pour la mise en œuvre du noyau du système distribué GOTHIC et d'un environnement de programmation adapté. Le but poursuivi est la conception d'un petit système de développement pouvant s'exécuter sur les versions successives du noyau GOTHIC. Le noyau est ainsi intensivement testé, et les erreurs de conception rapidement détectées.

Mots clés :

Systèmes distribués, Unix.

*GOTHIC projet INRIA-BULL

IMPLEMENTATION OF AN UNIX ENVIRONMENT ON THE GOTHIC* KERNEL.

Pascale LE CERTEN, Béatrice MICHEL
BULL RECHERCHE

Gilles MULLER
IRISA-INRIA
Campus de Beaulieu, 35042 Rennes-CEDEX

Abstract :

Creating an operating system on an open machine, implies the use of development methods. This report describes the way chosen to implement the kernel of the GOTHIC distributed system and a well suited programming environment. Our goal is to design a development system which can run on the successive versions of the kernel. The major advantage of that method is to intensively test the kernel for programming and design errors.

Keys words :

Distributed systems, Unix.

Résumé :

La réalisation d'un système d'exploitation sur une machine nue impose l'utilisation de méthodes de développement. Ce rapport décrit la démarche choisie pour la mise en œuvre du noyau du système distribué GOTHIC et d'un environnement de programmation adapté. Le but poursuivi est la conception d'un petit système de développement pouvant s'exécuter sur les versions successives du noyau GOTHIC. Le noyau est ainsi intensivement testé, et les erreurs de conception rapidement détectées.

Mots clés :

Systèmes distribués, Unix.

*GOTHIC projet INRIA-BULL

1- Introduction.

La réalisation d'un système d'exploitation sur une machine nue impose l'utilisation de méthodes de développement. De manière générale, l'écriture et la compilation des programmes sont effectuées sur un premier système d'exploitation dit hôte; l'exécution et le test se font sur la machine finale dite cible. Ce type de développement est appelé développement croisé.

Dans le cadre de la mise en œuvre du noyau du système distribué GOTHIC une démarche différente est utilisée. Le but de la méthode adoptée est de s'affranchir aussi rapidement que possible d'un système hôte. La mise en œuvre est effectuée de manière incrémentale, en intégrant au fur et à mesure de leur conception les différentes caractéristiques du noyau. Chaque version V_{i+1} du noyau est réalisée en utilisant la version V_i comme système de développement. Ceci est rendu possible en mettant à la disposition des concepteurs du système un environnement autonome de programmation et de test s'exécutant au dessus de la version initiale V_0.

Dans la section 2 de ce rapport, nous présentons un rappel sur le noyau GOTHIC. Le principe de la méthode et ses avantages sont exposés dans la section 3. La section 4 est dédiée au choix des caractéristiques de l'environnement. Les détails de la mise en œuvre sont présentés dans la section 5. La section 6 conclut le rapport.

2- Structure du noyau GOTHIC.

GOTHIC [BANA-86a] est un noyau de système distribué intégré tolérant aux fautes. L'outil de structuration proposé à cet effet est la multi-fonction [BANA-86b], qui permet la description de tâches parallèles imbriquées. Les multi-fonctions possèdent la propriété d'atomicité. De ce fait toutes les primitives du noyau (multi-fonctions) sont atomiques, ce qui évite la propagation des effets de bord en cas de panne tant au niveau du noyau, qu'au niveau utilisateur.

Les multi-fonctions résolvent deux des problèmes principaux du concepteur d'une application répartie et tolérante aux pannes :

- Les effets de bord sont limités en cas de panne du système,

- L'analyse globale de l'application est facilitée. Une multi-fonction remplace un ensemble de processus coopérants, ce qui facilite la compréhension et la maîtrise de la structure de contrôle.

Le noyau doit fournir une exécution atomique performante des multi-fonctions pour ne pas pénaliser les applications. Pour ce faire, les protocoles de mise en œuvre utilisent une Mémoire Rapide Stable (**MRS**) [BANA-87] qui offre des primitives d'accès atomiques (création, lecture, modification, destruction) aux objets stockés.

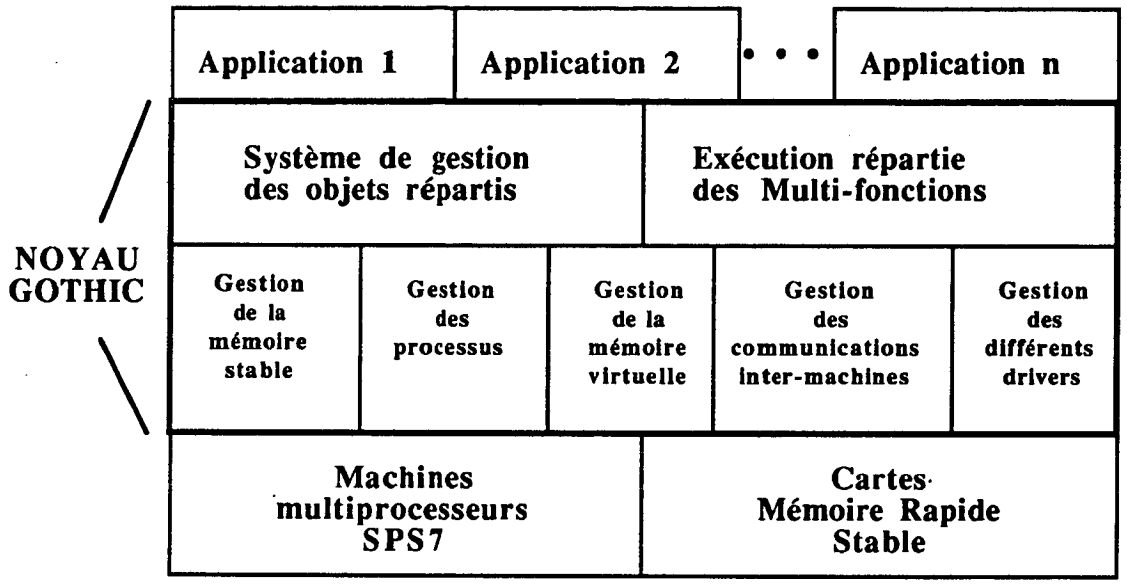

Figure 1: Structure du noyau GOTHIC.

Le noyau est composé des couches suivantes (cf. fig.1) :

- La machine de base est composée d'un ensemble de machines multiprocesseurs SPS7 [BULL-85] enrichies d'une carte MRS par processeur.

- Au dessus de la machine, une première couche logicielle comprend les différents modules de gestion du matériel (la mémoire stable MRS [COQU-87], les communications [LUCA-87], le disque, ...), et ceux associés à la gestion de la mémoire virtuelle et des processus. Cette couche est réalisée à partir d'un système temps réel SPART [BULL-86a]. L'interface de programmation est détaillée en annexe à la fin du rapport.

- Une deuxième couche répartie sur l'ensemble des machines prend en charge l'exécution

des multi-fonctions et la gestion des objets. Cette couche est actuellement en cours de spécification.

3- Principe de la méthode de développement.

Le noyau GOTHIC est développé en versions successives, au fur et à mesure de la mise en œuvre des couches le composant. Si l'écriture de la version initiale V_0 doit être effectuée sur un système distinct par un développement croisé, il est intéressant de réaliser la version V_{i+1} en utilisant la version précédente V_i comme système de développement. Cette méthode de travail impose une utilisation poussée de la version V_i et met rapidement en évidence les erreurs de programmation et de conception. Le système résultant n'est pas une maquette ou un "jouet", mais un système véritablement utilisable.

Pour ce faire, les programmeurs doivent avoir à leur disposition un environnement utilisateur qui leur permette de se connecter à la machine, d'écrire, de compiler et de tester leurs programmes. La mise en œuvre de cet environnement s'effectue sous la forme d'une application s'exécutant au dessus de la version courante V_i. L'ensemble "environnement" et "version V_i" constitue un système d'exploitation autonome exécutable sur machine nue. Cette autonomie permet de s'affranchir d'un système hôte aussi bien pour le développement que pour l'exécution. Les performances du noyau ne sont pas pénalisées par la présence d'une couche logicielle inférieure, et aucune contrainte ne pèse sur la conception et la mise en œuvre.

La version initiale V_0 est pratiquement réduite au système SPART et aux gestionnaires de la mémoire stable (MRS) et des communications. En conséquence, la conception de l'environnement de développement est effectuée en fonction de SPART, et de son système de gestion de fichiers. Cependant, cet environnement n'est pas figé et doit être modifié pour s'adapter aux évolutions de chaque version (multi-fonctions, objets). A terme celui-ci doit devenir une des applications GOTHIC.

4- Choix d'un environnement utilisateur.

Il existe deux possibilités pour le choix d'un environnement utilisateur. La première est de créer un environnement complet adapté aux concepts de GOTHIC (multi-fonctions, tolérance aux

pannes, ...) et utilisant les derniers raffinements en matière de communication homme-machine (Multifenêtrage, souris, ...). La deuxième consiste à adapter l'interface de programmation (bibliothèques) d'un système existant, puis à recompiler les outils et les commandes nécessaires en utilisant ces bibliothèques.

Le développement d'un gros projet tel que le noyau GOTHIC impose au minimum l'utilisation d'un gestionnaire de versions de programmes sources et de mise à jour des codes exécutables. Un environnement utilisateur minimal, mais adapté aux besoins des développeurs doit au moins satisfaire les conditions suivantes (cf. fig.2) :

- Possibilité de connection au système en assurant la protection de l'espace de travail de l'utilisateur.
- Disponibilité de commandes générales de manipulation de fichiers.
- Disponibilité d'une chaîne de compilation et d'un gestionnaire de versions de programmes sources.
- Disponibilité d'un éditeur de texte évolué.
- Disponibilité de quelques commandes de maintenance des disques et des espaces de travail utilisateurs.

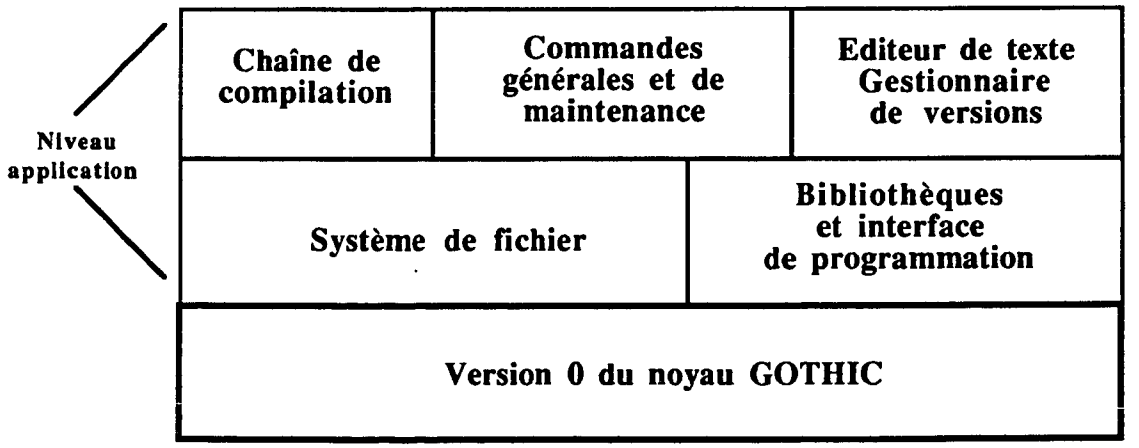

Figure 2: Structure de l'environnement.

La conception complète d'un environnement est nécessaire à terme pour mettre en valeur et

exploiter efficacement les concepts du système GOTHIC, mais impose une réécriture complète des outils cités ci-dessus. Ce développement est incompatible avec l'impératif de rapidité de disponibilité de l'environnement. De ce fait, il est préférable de choisir la solution du portage qui limite pratiquement le coût à la réécriture de l'interface de programmation et permet l'adjonction de nouveaux outils avec très peu de modifications.

Nous avons choisi de réaliser un environnement compatible avec le système d'exploitation UNIX System 5.2 (constructeur ATT) pour plusieurs raisons :

- Disponibilité des programmes sources. L'INRIA a acquis la licence ATT de System 5.2. Les outils et commandes à porter sont distribués avec UNIX ou sont dans le domaine public.

- Travail minimal à effectuer. Le noyau temps réel SPART est distribué avec un système de gestion de fichiers compatible UNIX (FMS) .

5- Mise en œuvre de l'environnement (Version 0).

5.1- Description du système SPART.

Le système d'exploitation SPART est un système permettant la mise au point d'applications "temps réel" sur l'architecture multiprocesseur SPS7. La définition de SPART s'appuie sur la méthodologie proposée dans le rapport SCEPTRE [BROW-84].

Ce système permet de gérer un à huit Modules de Traitement (MT). Les MT sont dédiés, c'est à dire que chaque module possède son propre noyau local. Une tâche (un processus SPART) est créée sur un module et y demeure jusqu'à la fin de son exécution. Toutes les requêtes concernant une tâche (création, activation, destruction, ...) sont locales à un MT. Par l'intermédiaire de la mémoire globale et du noyau de communication interprocesseur, SPART permet à des tâches s'exécutant sur des MT différents de partager de la mémoire commune, d'échanger des messages, de se synchroniser (sémaphores).

SPART est organisé autour d'un noyau qui gère l'ordonnancement de tâches, la signalisation élémentaire (événements), l'exclusion mutuelle de base (régions), les interruptions et les déroutements. En plus des fonctions de la norme SCEPTRE, le noyau permet la communication interprocesseur. Autour du noyau, on trouve un ensemble d'agences qui

offrent des services de plus haut niveau. L'ensemble de ces services constitue l'interface offerte aux applications, les fonctions de base du noyau n'étant pas directement accessibles.

L'entité de base du système SPART est l'objet (objet tâche, objet programme, objet segment...). Une agence permet la mise en œuvre d'une classe d'objets. Elle offre des primitives de création et de destruction d'objet de la classe, ainsi que les fonctions particulières à cette classe. Les différentes agences de base disponibles sont :

- Agence de gestion des tâches (processus SPART).
- Agence de gestion des programmes (objet contenant le code d'une tâche).
- Agence de gestion des segments (objet de base désignant une partie de la mémoire).
- Agence de gestion des boîtes aux lettres.
- Agence de gestion des sémaphores.
- Agence de gestion des délais.
- Agence de gestion des blocs d'entrées-sorties.

Les objets peuvent soit résider en mémoire locale, soit résider en mémoire globale. Dans le premier cas ils ne sont accessibles que par le MT qui les a créés, dans le second cas ils sont accessibles par tous les MT.

Le système SPART est livré avec une tâche opérateur qui permet aux utilisateurs de créer et de contrôler des tâches à partir de la console système, en appelant des primitives spécifiques. Ces primitives sont décrites dans l'annexe A.

5.2- Méthode de travail.

L'environnement utilisateur doit être adapté à la version du noyau au dessus duquel il s'exécute. Dans cette première version, le noyau est réduit au système SPART. De ce fait, les bibliothèques de bas niveau sont conçues avec une interface pour les objets SPART et devront être réécrites en termes de multi-fonctions, lorsque celles-ci auront été mises en œuvre. La version de base du système SPART servant au développement de l'environnement est la version V21.1. Cette version n'est pas la dernière version distribuée (V21.3), mais la seule dont nous possédons les sources. Une partie de la mise en œuvre est consacrée à l'adaptation de cette version à nos besoins.

Le développement de la première version de l'environnement doit être effectué de manière croisée. L'écriture et la compilation des programmes sont réalisées sur le système

SPIX[BULL-86b] (UNIX System 5.2 de BULL). La configuration matérielle de développement comprend une SPS7 bi-processeur, une unité de traitement étant dédiée à SPART, l'autre à SPIX. Les deux systèmes fonctionnent de manière complètement indépendante. Le partage de volume logique en écriture est interdit, car les mémoires caches de chaque système de fichier sont en mémoire locale. De ce fait, nous faisons communiquer les deux systèmes par l'utilisation d'un même volume en exclusion mutuelle (cf. fig.3). Les fichiers exécutables produits sous SPIX sont recopiés sur ce volume, le volume est démonté de SPIX, puis remonté sous SPART. Ceci permet de préserver la cohérence de chaque système de fichier.

Pour créer l'environnement cible, ce volume est monté, sous SPIX, sur un répertoire de nom "/gothic". Toutes les procédures de génération recopient les fichiers objets dans une arborescence similaire à celle d'UNIX à l'intérieur du volume. Sous SPART, celui-ci est monté comme racine du système de fichier et est directement utilisable par toutes les commandes transportées. La tâche opérateur permet d'effectuer toutes les opérations de montage et de démontage du volume partagé et de lancer l'exécution de l'environnement.

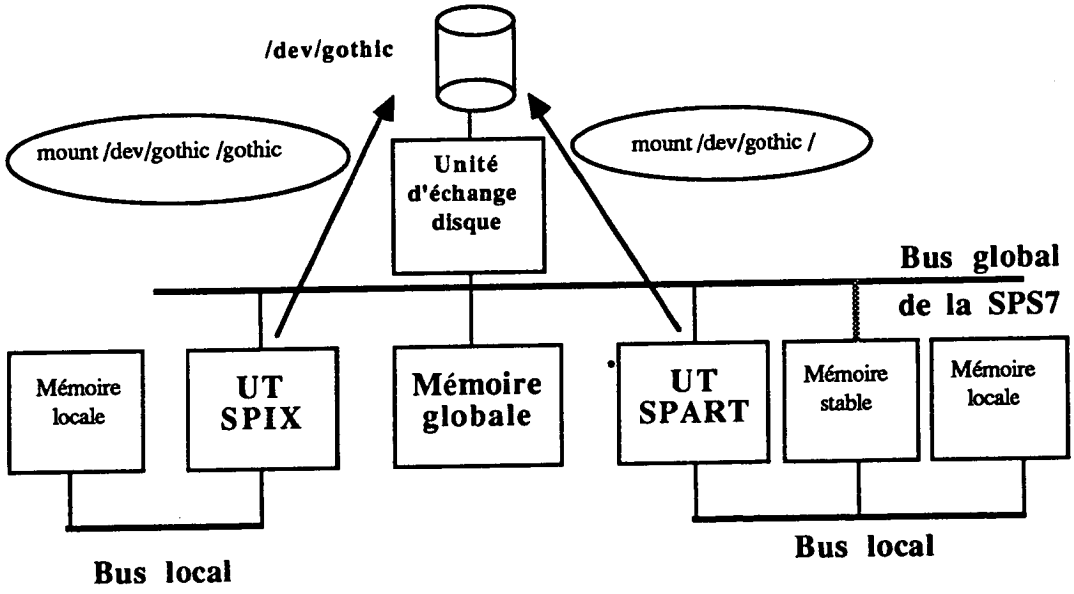

Figure 3: Configuration de développement

5.3- Description de la mise en œuvre.

Bien que l'environnement développé soit compatible System 5.2, une partie des sources utilisés provient d'une version V7, SMX 4.1 [GIPSI]. Ceci est dû, d'une part, au fait que ces

sources étaient les seuls dont nous disposions au début du portage, et d'autre part, à la dépendance de commandes, comme la chaîne de compilation, vis à vis du processeur (taille des mots, langage machine). Comme les sources System 5.2 sont livrés configurés pour le "VAX", nous avons préféré adapter la chaîne V7, plutôt que de réécrire une chaîne "68020". Ces raisons expliquent un certain mixage dans la provenance des sources transportés.

Nous avons fait le choix de ne pas mettre en œuvre les signaux dans la première version de l'environnement. Dans UNIX, les signaux sont généralement utilisés pour autoriser la destruction d'une tâche par l'utilisateur par appui sur des touches spécifiques. Cette fonction n'est réalisable qu'au prix de modifications dans les couches basses de SPART. Comme une tâche peut être toujours détruite par la console système, cet investissement a été jugé moins prioritaire que le reste de la mise en œuvre.

L'environnement est composé de deux couches (cf. fig.4). La première interface le noyau et effectue la gestion des objets UNIX, c'est à dire les processus et les fichiers. Cette couche est composée du système de gestion de fichiers FMS étendu, des bibliothèques de programmation (libc, ...) et des primitives UNIX de création de processus (fork, exec, ..). La deuxième couche contient l'ensemble des commandes et des outils accessibles à l'utilisateur. Le développement de l'environnement consiste à mettre en œuvre ces deux couches, soit par le transport de programmes existants, soit par la réécriture de primitives dépendantes de la machine.

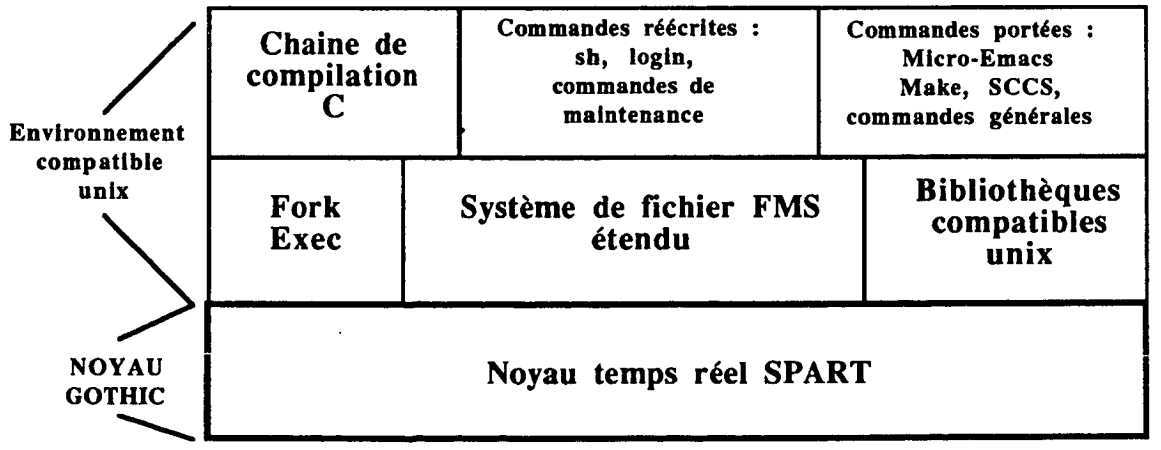

Figure 4: Environnement compatible UNIX.

5.3.1- Interface et bibliothèques de programmation.

La mise en œuvre de cette couche comporte trois travaux principaux : premièrement, le système de gestion de fichiers FMS livré (version V21.1) nécessite quelques adaptations pour correspondre exactement à nos besoins. Deuxièmement, la notion de processus étant différente dans UNIX et dans SPART, les primitives de gestion des processus doivent être réécrites au dessus de SPART. Enfin, les bibliothèques de primitives utilisées à l'édition de lien des commandes doivent être transportées.

La plus importante de ces bibliothèques, la "libc" qui contient les appels systèmes, est en partie fournie avec FMS. Le complément doit être réécrit ou porté depuis les sources de Système 5.2. Les autres bibliothèques ne comportent pas d'appels systèmes, et peuvent être transportées directement.

5.3.1.1- Modification de FMS.

Le système de fichier FMS est un sous-ensemble de celui de UNIX System 5.2. Certaines fonctionnalités comme les "pipes" sont absentes. De plus, son but initial est uniquement de pouvoir utiliser ou échanger des fichiers avec un système SPIX. Ceci explique la découverte de quelques erreurs et insuffisances, dans la gestion des voies d'entrée-sortie séries.

Le premier problème rencontré concerne la redirection des quatre fichiers standards d'entrée-sortie (0 à 3). Sous FMS, lors de la création d'un processus, ces fichiers sont affectés respectivement à la lecture console, l'écriture console, la sortie d'erreur et la sortie console système. Comme en standard, il n'y a qu'une console, les sorties 1 à 3 sont équivalentes.

Pour des raisons d'optimisation, et seulement pour ces quatre fichiers, les descripteurs logiques des fichiers au niveau de la tâche sont toujours ouverts, et les ordres de fermeture relatifs sont court-circuités et jamais exécutés (cf. fig.5). Ceci interdit, en outre, la redirection d'une entrée-sortie vers une console utilisateur (équivalente à une fermeture suivie d'une réouverture du même fichier). Le problème a été résolu en redonnant la possibilité de fermer ces quatre fichiers.

A chaque fermeture, le compteur du nombre d'ouvertures du fichier est décrémenté et le fichier est fermé au niveau de la tâche en mettant à zéro le descripteur logique. Lorsque le compteur passe à zéro, le fichier est fermé physiquement.

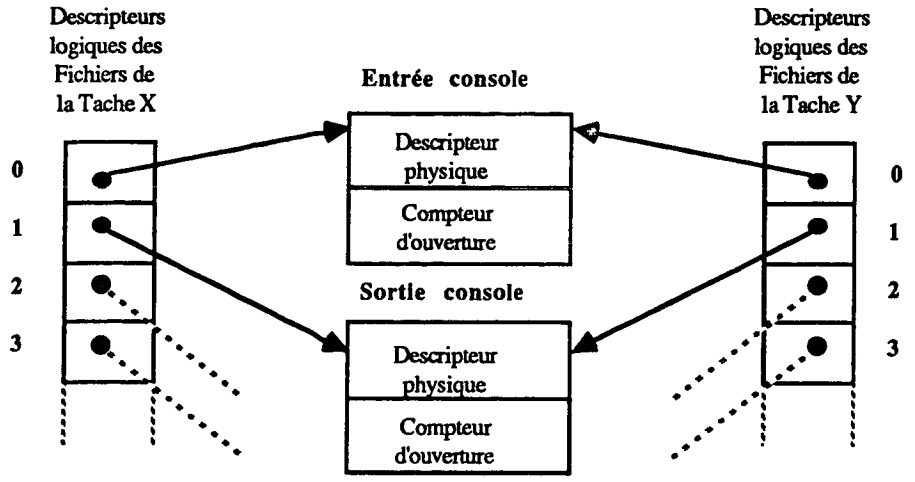

Figure 5: Fichiers standards de FMS.

De par ses buts initiaux, certaines fonctionnalités de gestion des voies d'entrée-sortie séries sont absentes dans FMS. Ces fonctions permettent de contrôler le comportement de l'interface vis à vis de certains caractères, comme le caractère d'effacement, de retour à la ligne, la présence d'un écho, la synchronisation par XON-XOFF, la vitesse de la ligne, etc... Dans System 5.2, les caractéristiques d'une voie sont regroupées dans une structure de nom "termio", cette structure peut être lue ou programmée par une commande "ioctl" associée respectivement aux fonctions "TCGETA" et "TCSETA" (cf. fig.6). Dans FMS, chaque commande de contrôle est accessible par une fonction "ioctl" spécifique.

Pour respecter l'interface "UNIX", les fonctions "TCGETA" et "TCSETA" ont été ajoutées aux commandes FMS standards, ainsi que des fonctionnalités comme le test de la frappe d'un caractère et l'émission d'un saut de ligne après chaque retour chariot.

Toute tâche FMS possède un répertoire courant. A la création de la tâche, ce répertoire est la racine. Lorsqu'un processus effectue un changement de répertoire, celui-ci est marqué "occupé". Dans la version SPART V21.1, à la terminaison du processus cette information n'est pas effacée. Aussi, si le répertoire se trouve sur un volume monté, ce volume n'est plus jamais démontable. Les programmes sources concernés par ce problème ont été identifiés et l'erreur a été corrigée.

UNIX

```
ioctl (desc_fichier, fonction, arguments)
    int desc_fichier;
    int fonction;
    struct termio *arguments;

fonctions :
    TCGETA      Lecture des caractéristiques
                associées à la voie série.
    TCSETA      Programmation des
                caractéristiques associées à
                la voie série.

struct termio {
    unsigned short c_iflag; /* input modes */
    unsigned short c_oflag; /* output modes */
    unsigned short c_cflag; /* control modes */
    unsigned short c_lflag; /* local modes */
    char  c_line ; /* line discipline */
    unsigned char c_cc[8]; /* control chars */
};
```

FMS

```
ioctl (desc_fichier, fonction, arguments)
    int desc_fichier;
    int fonction;
    struct termio *arguments;

fonctions (extraits):
    FDX       Passage de la voie en mode full-duplex.
    HDX       Passage de la voie en mode half-duplex.
    ECHO      Echo des caractères en réception.
    NO_ECHO   Suppression de l'écho.
    XON       Contrôle de flux par XON-XOFF.
    NO_XON    Supression du contrôle de flux.

Le format des arguments est dépendant de
la fonction.
```

Figure 6: Comparaison des commandes IOCTL.

Dans FMS, seul l'administrateur système (root) peut effectuer une création ou une destruction de lien et de répertoire. Cette restriction est évidemment incompatible avec un environnement utilisateur. Pour oter cette contrainte, le contrôle utilisateur="utilisateur système" existant avant ces opérations a été supprimé.

5.3.1.2- Gestion des processus UNIX sous SPART.

Un processus UNIX est caractérisé par un contexte d'exécution. Celui-ci contient un ensemble d'informations relatives au processus et au milieu dans lequel il s'exécute. Il contient les informations suivantes :

- uid, gid : Identificateurs réels d'utilisateur et de groupe du processus. Ces identificateurs sont hérités du processus père.

- euid, egid : Identificateurs effectifs d'utilisateur et de groupe du processus. Ceux-ci servent à déterminer les droits d'accès aux fichiers et sont hérités du processus père. Ces identificateurs sont égaux aux uid et gid, à moins qu'un changement de droits (par la primitive exec) ait été effectué par le processus ou un de ses ancêtres.

- Répertoire courant du processus.

- Ensemble des fichiers ouverts par le processus.

- umask : Masque sur les droits de création des fichiers.

- Paramètres d'appels "argc", "argv" de main pointant sur les arguments de la ligne de commande.

- Un environnement UNIX: il est caractérisé par un ensemble de variables. Chacune d'elles est associée à une chaîne de caractères. Ceci permet de définir par exemple le type du terminal (variable TERM), les règles de recherche des commandes exécutables (variable PATH). Par l'intermédiaire du "shell" (processus de commande), l'utilisateur peut étendre et consulter l'environnement. Pour le processus l'environnement est un tableau de chaînes de caractères pointé par la variable "environ".

Le contexte UNIX est mis en œuvre par un ensemble de structures de données. Le contexte FMS de la tâche constitue l'une d'entre elles. Il contient les différents identificateurs (uid, gid, euid, egid), le répertoire courant, les descripteurs de fichiers ouverts et la variable umask. L'environnement UNIX et les paramètres d'appel (équivalents à la ligne de commande) sont représentés par deux structures, conservées dans un segment associé au processus. Lors de la création d'un processus fils, ce segment est dupliqué pour que le fils hérite du contexte du père.

La primitive de création d'un processus UNIX est le "fork" (cf. fig.7). Cette primitive duplique un processus de manière analogue à une division cellulaire. En conséquence les deux processus résultants possèdent un code commun, et une pile d'exécution, un segment de donnée, et un contexte équivalents. Au retour de l'appel, la primitive rend à un des processus (le "fils"), la valeur 0, et à l'autre (le "père"), le nom système du processus fils.

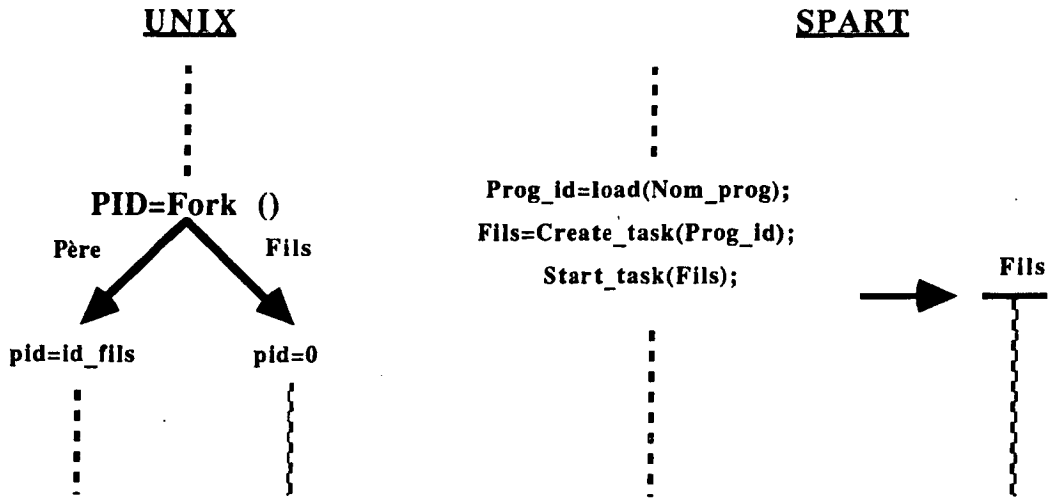

Figure 7: Comparaison de la création de processus.

Dans SPART un processus (tâche) est créé par une primitive "create_task" qui opère sur un objet "programme", préalablement chargé en mémoire. La tâche créée est en attente et doit être activée explicitement par la tâche mère. La tâche créatrice et la tâche créée n'ont aucun lien entre elles. La sémantique de SPART est donc complètement différente de celle d'UNIX.

La primitive "fork" n'est pas suffisante pour construire un système d'exploitation, car elle ne permet pas d'exécuter un processus avec un autre code que celui du père. Les concepteurs du système UNIX ont remédié à cela en introduisant une primitive "exec" qui, tout en conservant le contexte d'un processus, change de code, de données et de pile d'éxécution. La primitive "exec" est une primitive générique: elle existe sous plusieurs formes qui diffèrent par leurs paramètres d'appel.

A la création d'une tâche, FMS initialise les identificateurs (réels et effectifs) de groupe et d'utilisateur (uid, gid, euid et egid) avec des valeurs prédéfinies à la génération de FMS. Toutes les tâches ont, de ce fait, le même uid et gid. Dans un environnement multi-utilisateurs, on doit pouvoir positionner ces variables avec le numéro d'utilisateur et de groupe adéquat. Ceci oblige à initialiser les valeurs prédéfinies de FMS à la valeur de ceux de l'administrateur système, pour pouvoir ensuite, au début de la tâche, mettre à jour ces identificateurs avec la valeur désirée.

Deux mises en œuvre du "fork" et de l'"exec" sont possibles. Soit au dessus de FMS, en utilisant l'interface utilisateur normale, soit au même niveau que FMS, c'est à dire sous la forme d'une agence, donc du noyau SPART. La première solution présente l'avantage de la

facilité car seules les primitives utilisateurs sont utilisées, mais oblige à initialiser les identificateurs avec ceux de l'administrateur système. La protection des utilisateurs n'est pas alors assurée puisque la mise à jour des indentificateurs est effectuée par le processus même, et qu'il peut prendre les droits de quiconque. L'autre solution impose de travailler au niveau du noyau avec des objets de bas niveau, mais permet d'assurer la protection nécessaire. La mise en œuvre est plus difficile car toute erreur a pour effet de mettre en panne SPART, oblige à une recompilation du noyau et à un rechargement des deux systèmes (SPIX et SPART).

Processus père	Processus fils
Appel de Fork.	
- Incrémentation du compteur d'utilisation du programme contenant le code.	
- Création d'un objet tâche.	
- Allocation d'un bloc mémoire pour le contexte SPART de la nouvelle tâche.	
- Création d'un segment pour les données.	
- Création d'un segment pour la pile.	
- Création d'un segment pour l'environnement.	
- Initialisation du contexte SPART de la nouvelle tâche.	
- Démarrage de la nouvelle tâche qui prend la main.	
	- Recopie du segment de pile de la tâche mère dans celui de la tâche fille.
	- Recopie du segment de données.
	- Recopie du segment de l'environnement.
	- Accès au segment de code dans l'espace logique de la tâche.
	- Recopie des descripteurs de fichiers.
	- Recopie des identificateurs de l'utilisateur et du groupe.
	- Initialisation du répertoire courant.
	- Recopie de la variable umask.
	- Commutation de contexte.
	Retour avec la valeur 0.
.....	
Retour avec l'identificateur du processus fils.	

Figure 8: Mise en œuvre de la primitive fork.

Dans les faits, les deux solutions ont été chronologiquement réalisées. La première l'a été pour pouvoir disposer rapidement d'une création de tâche compatible UNIX. Cette mise en œuvre est un enchaînement du "fork" et de l'"exec" et a été seulement intégrée dans un mini-shell écrit pour tester rapidement des commandes UNIX. La seconde solution (cf. fig.8) a été réalisée pour disposer réellement des primitives "fork" et "exec", avec la protection adéquate.

Un processus UNIX qui a créé un fils peut éventuellement attendre que le fils se termine et que celui-ci lui retourne un compte rendu d'exécution. Ceci est notamment le cas de l'enchaîneur de passes "cc", qui appelle plusieurs programmes pour effectuer une compilation. Une passe n'est exécutée que si la précédente a rendu un code de retour correct. La primitive UNIX d'attente est "int wait ()", celle de terminaison de processus est "exit (code)".

La mise en œuvre du compte rendu est réalisée en créant une variable "CR" dans le segment environnement du processus fils. Lorsque le processus fils exécute un "exit (code)", la variable "CR" est affectée avec la valeur "code". La mise en œuvre du "wait ()" consiste à accéder au segment environnement du dernier fils créé, puis à attendre la fin du processus par la primitive SPART "wait_end_task". Au retour de cette procédure, le père peut récupérer le compte rendu et relacher l'accès au segment environnement du fils. Sous SPART, la destruction d'un segment environnement n'est effective que lorsque le dernier accès en cours est terminé. De ce fait, le fils demande toujours la destruction de son segment environnement. La destruction est validée lorsque le père relache l'accès, si celui-ci a appelé "wait ()", ou immédiatement, s'il n'y a pas d'attente en cours.

Une possibilité offerte par la primitive "exec" est de pouvoir changer les identificateurs d'utilisateur du processus. Si le bit "S" est positionné à la place du droit "x", pour le propriétaire ou le groupe, sur le fichier contenant le code, le processus s'exécute avec les identificateurs effectifs de propriétaire ou de groupe (uid et gid) du fichier (cf. fig.9). Les identificateurs réels restent ceux du processus appelant. Ceci permet de mettre en œuvre des commandes qui doivent avoir un niveau supérieur de visibilité du système, comme "su" (passage en administrateur du système), ou qui doivent avoir accès à un domaine réservé, comme le spooler d'impression "lp".

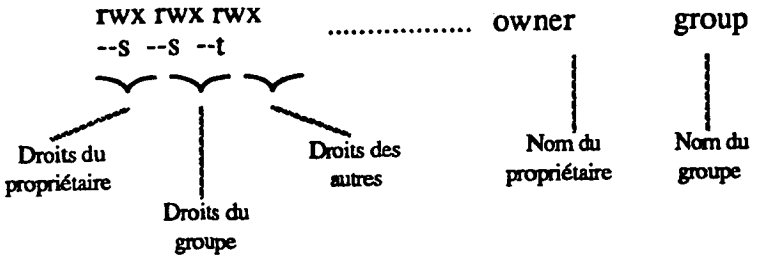

Figure 9: Droits d'accès sur les fichiers.

La mise en œuvre de la primitive "exec", consiste à ne garder que le contexte du processus appelant et à réinitialiser ensuite les segments de code, de donnée, de pile à partir d'un fichier exécutable. Ces différents segments peuvent être de tailles différentes dans les deux processus. Ceci est valable pour la pile, qui doit être allouée la première en bas de l'espace logique, afin de détecter les débordements à l'exécution. Le segment d'environnement, qui est transmis, n'est donc pas forcément réalloué à la même adresse (cf. fig.10). Celui-ci contient un nombre fixe de pointeurs (environnement et ligne de commande) vers des variables internes, qui ne sont plus corrects après le changement d'adresse d'allocation, et qui doivent être convertis. Un pointeur peut être représenté par un couple dont le premier élément (base) est l'adresse d'allocation du segment environnement, et le second, un déplacement dans le segment. La fonction de conversion est donc :

$$Point_{nouv} = Point_{anc} - Base_{ancienne} + Base_{nouvelle}.$$

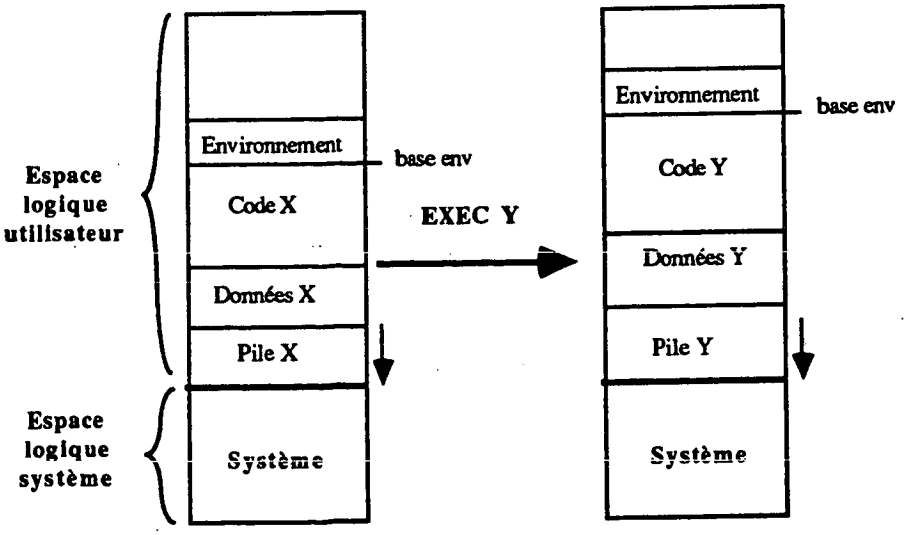

Figure 10: Adresse d'allocation des segments après un Exec.

Une solution simplifiée de la mise en œuvre de "exec" est explicitée en figure 11.

Appel de exec.

- Vérification de l'accès et des droits d'exécution du nouveau fichier.
- désallocation du segment ENVIRONNEMENT.
- désallocation du segment de CODE.
- désallocation du segment de DONNEE.
- désallocation du segment de PILE.
- Décrémentation du compteur d'utilisation de l'ancien programme.
- Chargement du nouveau programme à partir du fichier.
- Incrémentation du compteur d'utilisation du nouveau programme.
- Destruction du nouveau programme. Cet ordre ne sera exécuté que lorsque plus aucune tâche ne l'utilisera.
- Mise à jour des identificateurs effectifs si un bit "S" est positionné sur le fichier.
- Allocation du segment de PILE.
- Allocation du segment de DONNEE.
- Allocation du segment de CODE.
- Allocation du segment ENVIRONNEMENT.
- Conversion des pointeurs du segment ENVIRONNEMENT.

Retour.

Figure 11: Mise en œuvre simplifiée de la primitive exec.

5.3.1.3- Transport des bibliothèques de programmation.

Le transport de la bibliothèque "libc" ne nécessite qu'une recompilation de la plupart des primitives. Cependant les fonctions dépendant de la machine ou de la gestion des processus ont été réécrites.

Bien que les signaux ne soient pas mis en œuvre, des primitives vides ont été réalisées pour ne pas créer d'erreurs à l'édition de lien. Il est à noter que l'utilisation la plus générale de ces primitives est d'interdire la destruction de la tâche à des moments risquant de rendre incohérent le système.

La lecture du temps est effectuée par une fonction "time" qui rend un entier indiquant le nombre de secondes écoulées depuis le 1er janvier 1970. Sous FMS, cette fonction n'est pas

disponible sous cette forme, mais existe cependant. Elle a été utilisée pour réécrire une primitive "time".

Au lancement d'un processus, la première instruction exécutée n'est pas la procédure "main", mais une procédure "startup" fournie par le système. Le rôle de cette procédure est évidemment d'appeler "main", mais aussi d'effectuer l'initialisation du tas, des fichiers standards, et de l'environnement UNIX. La gestion des processus UNIX ayant été réécrite au dessus de SPART, la fonction "startup" a été modifiée en conséquence.

Dans le système, les tâches SPART "pures" et les processus UNIX cohabitent. De ce fait, les deux formes de la fonction "startup" existent dans deux bibliothèques. A l'édition de lien, une des deux bibliothèques est utilisée suivant le type de la tâche.

Pour faciliter l'accès aux variables de l'environnement UNIX, il existe des primitives de consultation "getenv" et d'écriture "putenv". Ces fonctions sont nécessaires et ont été réécrites.

En dehors de la "libc", nous avons transporté deux autres bibliothèques utilisées par des commandes:

- Bibliothèque de fonctions de gestion des terminaux "libcurses". Ces primitives sont utilisées par l'éditeur de texte et permettent l'emploi de consoles différentes.

-Bibliothèque de gestion du format des codes exécutables "libld". Ces primitives permettent l'accès aux entêtes et structures des programmes objets et sont utilisées par les outils System 5.2 de gestion des codes exécutables (load, dump, convert ...).

5.3.2- Commandes utilisateur.

Nous distinguons les commandes réécrites et les commandes portées à partir de sources UNIX V7 et UNIX system 5.2 disponibles.

5.3.2.1- Commandes spécifiques ou réécrites.

Le lancement de l'environnement est effectué par une tâche "init", créée par l'opérateur au moyen de la console système. Cette tâche attend le "login" d'un usager sur une seule console utilisateur. En cas de succès, un processus de commande "shell ou sh" est créé pour cet utilisateur. Dans la configuration actuelle, deux tâches "init" sont exécutées, ce qui permet l'accès du système à deux usagers.

Le shell ("sh") est le processus qui permet à l'utilisateur de taper et d'exécuter des commandes. Notre environnement étant un sous-ensemble d'UNIX, nous avons préféré réécrire un mini-shell n'utilisant que les fonctionnalités d'UNIX mises en œuvre. Une raison historique à la conception de ce shell est la possibilité de pouvoir tester des commandes alors que les primitives "fork" et "exec" n'étaient pas mises en œuvre.

5.3.2.2- Environnement de programmation.

Un certain nombre de commandes de base (df, du, ls, pr, cat, mkdir, rm, ...) ont été portées en adaptant les sources de UNIX V7 dont nous disposions. Les autres commandes portées proviennent de UNIX system 5.2.

La chaîne de compilation du langage C est un composé hybride de source V7 et System 5.2. La chaîne est composée d'un enchaîneur de passe "cc" qui appelle successivement le préprocesseur "cpp", le compilateur "c0", l'optimiseur "c2", l'assembleur "as" et l'éditeur de lien "ld". Les sources System 5.2 sont livrés configurés pour un "VAX". Aussi, pour ne pas avoir à réécrire le compilateur et l'assembleur pour un "68020", nous reprenons les premières passes (de "cpp" jusqu'à l'assembleur) des sources V7 qui étaient écrits pour un "68000". Ensuite nous convertissons le programme objet au format System 5.2 (commande "convert") et utilisons l'éditeur de lien de System 5.2. Tous les autres outils de gestion des codes objets (load, dump, size, s3 ...) sont également repris de System 5.2.

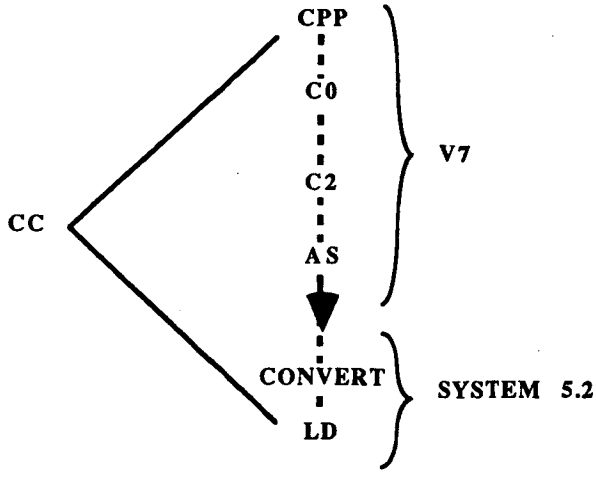

Figure 12: Structure de la chaîne de compilation C.

Les outils "make" et "sccs" sont utilisés pour gérer respectivement les codes objets et les programmes sources. L'éditeur mis à la disposition des utilisateurs est une version réduite d'emacs disponible dans le domaine public (MICRO-EMACS 3.7). Cet éditeur est suffisamment puissant pour donner une interface conviviale, tout en restant de taille raisonnable et de portage facile.

6- Conclusion.

Nous disposons à présent d'un environnement de programmation UNIX mis en œuvre au dessus du noyau primitif du système GOTHIC, c'est à dire SPART étendu avec la mémoire stable et les communications (version V_0). Les prochaines versions du système seront obtenues directement à partir de cette version initiale, en utilisant l'environnement de travail construit. Nous sommes ainsi assurés d'obtenir une version finale du système largement testée.

Lorsque la couche logicielle prenant en charge l'exécution des multi-fonctions et la gestion des objets sera disponible , nous envisageons de réécrire l'environnement de programmation et le système de gestion de fichiers en termes d'objets GOTHIC.

Références:

[BANA-86a] BANATRE J.P., BANATRE M., PLOYETTE Fl.
An overview of the GOTHIC Distributed Operating System.

[BANA-86b] BANATRE J.P., BANATRE M., PLOYETTE Fl.
The Concept of Multi-fonction: a General Structuring tool for Distributed Operating System.
Proc. of 6th Distributed Computing Systems, Cambridge, Mass., Mai 1986.

[BULL-85] BULL (Cie).
Structure générale de la SPS7.
Rapport technique, BULL, Janvier 1986.

[BULL-86a] BULL (Cie).
SPART, Système d'exploitation temps réel Version 21.
Manuel de référence, BULL, Mai 1986.

[BULL-86b] BULL(Cie).
SPIX, Présentation générale.
Manuel de référence, BULL, Avril 1986.

[COQU-87] COQUELIN D.
Intégration d'une mémoire stable dans une architecture multi-processeurs.
Rapport de fin d'études D.E.S.S. I.S.A., I.F.S.I.C. Université de RENNES 1. Juin 1986.

[LUCA-87] LUCAS C.
Développement d'un système de communication pour réseau de machines SPS7.
Rapport de fin d'études I.N.S.A., Juin 1986.

[BANA-87] BANATRE J.P., BANATRE M., MULLER G.
Ensuring Data Security and Integrity with a fast stable storage.
Proc. of 4th on Data Engineering, Los Angeles Febr 88 (to appear).

[BROW-84] BROWAEYS F., DERRIENNIC H., DESCLAUD P., FALLOUR H., FAULLE C., FEBVRE J., HANNE J.E., KRONENTAL M., SIMON J.J., VOJNOVIC D.
SCEPTRE: proposition de noyau normalisé pour les exécutifs temps réel.
TSI Volume 3 n°1, janvier 1984, pp 45-62.

ANNEXE A

Primitives utilisateurs du système temps réel SPART.

Status **Access_segment** (segment_id, acces_type, logical_base_address).

Accès à un segment et allocation d'un espace logique dans l'espace de la tâche.

Status **Alloc_memory** (segment_id, block_length, block_adress).

Allocation d'un bloc de mémoire dans un segment.

Status **Catalog** (object_id, object_name).

Rangement d'un objet dans le catalogue local ou global.

Status **Create_mbx** (attribute, message_nb, mbx_id).

Création d'un boîte aux lettres locale ou globale.

Status **Create_segment** (attribute, length, segment_id).

Création d'un segment en mémoire locale ou globale.

Status **Create_semaphore** (attribute, semaphore_id).

Création d'un sémaphore local ou global.

Status **Create_task** (program_id, priority, privilege, max_resume, task_id).

Création d'une tache.

Status **Cycle_resume** (task_id, period, delay_id).

Demande de réactivation cyclique d'une tâche.

Status **Deaccess_segment** (segment_id).

Fin d'accès à un segment et libération de l'espace logique correspondant de la tâche appelante.

Status **Declare_segment** (segment_id, granule_length).

Initialisation d'un segment pour utilisation des algorithmes d'allocation et de libération de blocs.

Status **Delayed_event** (event_num, task_id, delay, delay_id)

Signalisation différée d'un évènement à une tâche.

Status **Delayed_message** (message, mbx_id, delay, delay_id).

Envoi différé d'un message sur une boîte aux lettres.

Status **Delayed_resume** (task_id, delay, delay_id).

Réactivation différée d'une tâche.

Status **Delete_mbx** (mbx_id).

Supression d'une boîte aux lettres.

Status **Delete_segment** (segment_id).

Destruction d'un segment et libération de l'espace physique correspondant.

Status **Delete_semaphore** (semaphore_id).

Suppression d'un sémaphore.

Status **E_request** (semaphore_id, delay).

Demande d'accès exclusif à un sémaphore.

Status **Ex_connect** (handler).

Connection d'un handler d'exception.

Status **Exec_task** (task_id, argv).

Démarrage d'une tâche avec passage de paramètres dans la pile.

Status **Exit_task** (return_value).

Fin normale d'une tâche.

Status **Find** (attribute, object_name, object_id).

Recherche d'un objet dans le catalogue local ou global.

Status **Free_memory** (segment_id, block_address, block_length).

Libération d'un bloc de mémoire.

Status **Ftime** (time_p).

Lecture de l'heure système exprimée en millisecondes depuis le 01/01/1970 à 0 heure GMT.

Status **Gen_addr** (logical_addr, general_addr).

Transforme une adresse logique locale à la tâche en une adresse généralisée (segment_id, déplacement).

Status **Get_task_param** (task_id, task_parameters).

Rend les paramètres de la tâche.

Status **Get_time** (current_time).

Lecture de l'heure système en millisecondes modulo 2 puissance 32 depuis le 01/01/1970 à 0h GMT (soit une période d 49 jours environ).

Status **Halt_io** (iocb_id).

Arrêt d'une entrée/sortie en cours.

Status **Io_access** (ppn, p_num).

Accès à un périphérique désigné par son PPN (Physical Peripheral Name).

Status **Io_deaccess** (p_num).

Fin d'accès à un périphérique.

Status **Kill_delay** (delay_id).

Interruption d'un délai en cours.

Status **Kill_task** (task_id).

Meurtre d'une tâche.

Status **Load** (unit_num, file_name, prog_type, prog_id).

Chargement d'une image mémoire exécutable.

Status **Loc_addr** (general_addr, logical_addr).

Transforme une adresse généralisée (segment_id, déplacement) en une adresse logique locale à la tâche.

Status **Map_segment** (segment_id, access_type, logical_base_addresss).

Mappe un segment dans l'espace logique à une adresse imposée par l'utilisateur.

Status **Offset** (logical_addr, segment_offset).

Rend le déplacement par rapport au début du sement, d'une adresse logique locale à une tâche.

Status **Receive** (message, mbx_id, delay).

Réception d'un message.

Status **Release** (semaphore_id).

Libération d'un sémaphore.

Status **Reset_event_list** (event_list).

Effacement d'un liste d'événements.

Status **Resume_task** (task_id).

Réactivation d'une tâche suspendue.

Status **Send** (message, mbx_id, delay).

Envoi d'un message de 8 octets sur une boîte aux lettres.

Status **Set_event** (event_num, task_id).

Signalisation d'un événement.

Status **Sleep** (delay).

Attente d'un délai pur.

Status **S_request** (semaphore_id, delay).

Demande d'accès partagé à un sémaphore.

Status **Start_io** (iopb, delay, iocb_id).

Lancement d'une entrée/sortie.

Status **Start_task** (task_id, message).

Activation d'une tâche.

Status **Stime** (time_p).

Modification de l'heure machine avec resynchronisation de l'horloge.

Status **Stop** (level).

Mise à l'arrêt d'un MT ou de l'ensemble des MT de la machine. STOP provoque un retour sous pupitre.

Status **Suspend_task** ().

Attente de réactivation.

Status **Sync_end_task** (task_id, mbx_id, user_id).

Demande de synchronisation par message sur fin de tâche.

Status **Test_event** (event_num, event_state).

Test de l'état d'un événement.

Status **Uncatalog** (attribute, object_name).

Sortie d'un objet du catalogue.

Status **Unload** (prog_id).

Libération d'un programme.

Status **Wait** (event_list).

Attente d'un "OU" d'événements désigné par une liste.

Status **Wait_end_task** (task_id, delay).

Attente de la fin d'une tâche.

Status **Wait_io** (iocb_id, delay, io_message).

Attente d'une fin d'entrée/sortie.

ANNEXE B

Primitives d'utilisation de la mémoire stable.

a- Les primitives du gestionnaire de mémoire stable

status **Create_stable_segment** (length, segment_id).

Création d'un segment en mémoire stable.

status **Delete_stable_segment** (segment_id).

Destruction d'un segment et libération de l'espace physique correspondant.

status **Access_stable_segment** (segment_id,access_type,logical_base_adress).

Accès à un segment stable et mapping dans l'espace logique de la tâche appelante. L'adresse logique à laquelle le segmer sera "vu" par la tâche est déterminée par le système.

status **Map_stable_segment** (segment_id,access_type,logical_base_adress).

Accès à un segment stable et mapping dans l'espace logique de la tâche appelante. L'adresse logique à laquelle le segmer sera "vu" par la tâche est fixe et déterminée par l'utilisateur.

status **Deac_stable_segment** (segment_id).

Fin d'accès à un segment stable et libération de l'espace logique correspondant de la tâche appelante.

status **Access_stable_registre** (logical_base_adress).

Accès aux registres du bus local de la mémoire stable et mapping dans l'espace logique de la tâche appelante.

status **Deac_stable_registre** ().

Fin d'accès aux registres du bus local de la mémoire stable et libération de l'espace logique correspondant de la tâch appelante.

int **Masque_ms** ().

Positionnement d'un verrou d'accès à la mémoire stable. Cette fonction doit être appelée avant chaque traitemen atomique d'objet.

int **Demasque_ms** ().

Libération du verrou d'accès à la mémoire stable. Cette fonction doit être appelée après chaque traitement atomiqu d'objet.

b-Les primitives atomiques de gestion d'objets stables.

status **Create_object** (topo_objet,adr_reg_stable).

Création d'un objet dans un segment stable réservé par l'utilisateur.

status **Create_sub_object** (topo_ss_obj,adr_reg_stable).

Création de tous les sous-objets d'un même niveau d'un objet précédemment créé.

status **Delete_object** (topo_objet, niveau, adr_reg_stable).

Destruction d'un objet précédemment créé.

status **Read_object** (topo_objet, topo_ram,niveau, adr_reg_stable).

Lecture d'un objet.

status **Update_object** (topo_objet, topo_ram,niveau, cle_authent, adr_reg_stable).

Mise à jour atomique d'un objet.

status **Quick_update_object** (topo_objet, topo_ram,niveau, cle_authent, adr_reg_stable).

Mise à jour rapide d'un objet, la phase de reconnaissance est supprimée.

status **Update_group_object** (liste_topo_objet, adr_reg_stable).

Mise à jour d'un groupe d'objets.

status **Quick_update_group_object** (liste_topo_objet, adr_reg_stable).

Mise à jour rapide d'un groupe d'objets.

ANNEXE C

Primitives de communication.

a-Les primitives de bas niveau de gestion du réseau.

filedesc **Alloc_exchange** ().

Allocation d'un descripteur de communication avec le gestionnaire de réseau.

status **Close_exchange** (filedesc).

Fermeture d'un descripteur de communication.

status **Select_com** (filedesc, site_tb, ecb).

Sélection d'un canal de communication (destinataire).

status **Close_com** (filedesc, ecb).

Fermeture d'un canal de communication (destinataire).

status **Send_frame** (filedesc, mes, length, ecb).

Envoi d'une trame sur le réseau.

status **Receive_frame** (filedesc, time_out, mes, length, ecb).

Réception d'une trame.

status **Read_conf** (filedesc, conf_tb).

Lecture de la table de configuration.

status **Write_conf** (filedesc, conf_tb, ecb).

Modification de la table de configuration.

b-Les primitives de communication multi-machines.

status **Send_message** (message, length, mbx, site).

Envoi d'un message sur une boîte aux lettres située sur un site quelconque du réseau.

status **Receive_message** (mbx, message, length).

Réception d'un message sur une boîte aux lettres.